From Learning 20 Days/Day to 10 Years/Day

(And Why NVIDIA is so rich)

Presented by AI/ML@TU

November 7, 2025

The Two Halves of Reinforcement Learning

A high-level look at Deep Q-Networks (DQN) / Deep RL:

- **Goal:** Train an "Agent" (a neural network) to make optimal decisions.
- **Two Key Jobs:**
  - **Data Collection (Acting):** The Agent (Policy) interacts with the Environment to gather experiences (`state`, `action`, `reward`, `next_state`).
  - **Model Training (Learning):** The Agent (Learner) samples batches of old experiences from a "Replay Buffer" to update its neural network.

Figure: The standard RL loop: Acting and Learning.

Pipeline 0: The "Basic" Gymnasium Loop (1x Baseline)

This is the code in most tutorials (and our competition last semester)

```
model = Model().to("cuda")
buffer = ReplayBuffer()
env = gym.make("CartPole-v1")
for episode in range(1000):
    state, _ = env.reset()
    while True:
        # 1. CPU runs one step
        action = model.act(state)
        next_state, reward, done, _, _ = env.step(action)
        # 2. CPU adds to buffer
        buffer.add(state, action, reward, next_state, done)
        # 3. CPU samples and...
        if len(buffer) > BATCH_SIZE:
            # 4. ...sends data from RAM to VRAM
            batch = buffer.sample()
            # 5. GPU trains for one step
            model.train(batch)
        if done: break

```

The Core Bottleneck: The "Data Moat"

To fix the problem, we must understand the hardware.

- **CPU (Central Processing Unit):** The "Manager." Great at serial logic. Uses **RAM** (System Memory).
- **GPU (Graphics Processing Unit):** The "Factory." Thousands of simple cores. Perfect for parallel math (neural nets). Uses **VRAM** (Video RAM).

The Bottleneck: RAM vs. VRAM

Data must move from RAM (where the CPU puts it) to VRAM (where the GPU needs it). This transfer over the **PCIe bus** is one of the **slowest operations** in modern computing.

Our Goal: Keep both CPU and GPU busy, and *minimize* RAM-to-VRAM transfers. (Draw stuff Timmy)

Solution Strategy 1: Parallelism (Multi-Threading)

- **Problem:** We have two jobs (Collection & Training) but one worker.
- **Solution: Multi-Threading!** Hire two workers (threads) to work at the same time.

- **Thread 1: The "CPU Worker"**
  - **Job:** *Only* runs the environment.
  - Constantly collects experiences.
  - Puts data into a buffer in **RAM**.

- **Thread 2: The "GPU Worker"**
  - **Job:** *Only* trains the model.
  - Constantly samples data from a buffer in **VRAM**.
  - Runs the `model.update()` loop as fast as it can.

New Problem: How to efficiently move data from the CPU buffer to the GPU buffer without stalling?

Pipeline 1: The "Double-Buffered" Pipeline

This pipeline decouples the "actors" from the "learner."

- **GPU Worker (Thread 1):**
  - Has the "main" Model and Replay Buffer on the **GPU**.
  - Runs the training loop constantly.
- **CPU Worker (Thread 2):**
  - Has a *copy* of the model on the **CPU**.
  - Fills a *local* buffer on the **CPU**.
- **The "Sync" (How they talk):**
  - **Data Transfer:** Every N steps, the CPU worker copies its *entire* local buffer (from **pinned memory** for faster transfer) as one big chunk from RAM to the GPU Buffer.
  - **Model Update:** Occasionally, the GPU worker copies its new, updated model weights to the CPU worker.
- **The "Double-Buffer" Trick:**
  - To minimize the CPU worker's "lock time" during model updates, we have *three* model copies (1 GPU, 2 CPU).
  - While the GPU updates CPU-model-A, the CPU worker instantly switches to using CPU-model-B via a pointer swap.

Pipeline 1 is Good... But the CPU is Still the Bottleneck

- **Problem:** Pipeline 1 is much faster, but our CPU worker is still slow. Why?
  - **Python is slow.** The `env.step()` loop, the `if/else` logic, all of it.
  - We are still only running **one environment**.

- **Solution:**
  - Run the environment code in a *much* faster language, like **C++**.
  - Don't run 1 environment. Run **768 environments** in parallel.

Pipeline 1 is Good... But the CPU is Still the Bottleneck

- **Problem:** Pipeline 1 is much faster, but our CPU worker is still slow. Why?
  - **Python is slow.** The `env.step()` loop, the `if/else` logic, all of it.
  - We are still only running **one environment**.

- **Solution:**
  - Run the environment code in a *much* faster language, like **C++**.
  - Don't run 1 environment. Run **768 environments** in parallel.

This leads to Pipeline 2. But first, some new concepts...

Concepts for Pipeline 2 (Part A): Bindings & Pointers

Concept: Language Bindings (e.g., Pybind11)

- **The "Glue."** We want C++ speed, Python control.
- Bindings let Python code call C++ functions *as if they were Python*.

```python
# Standard old Python
def my_function_old(x, y, z):
    # Do stuff
    # ...
    a, b, c = x + 1, y + 1, z + 1
    return a, b, c

# More Modern Python / Cython
def my_function_new(x: int, y: int, z: int) -> list[int]:
    # Do stuff
    # ...
    a, b, c = x + 1, y + 1, z + 1
    return [a, b, c]
```

Concepts for Pipeline 2 (Part B): Bindings & Pointers

Concept: Pointers (and Zero-Copy)

How do we get data from C++ to Python *without copying it*?

- **Normal (Python):** `my_data = [1, 2, 3]`. This variable *is* the data.
- **Pointer (C++):** `my_ptr = 0x7FFF1234`. This variable is just an *address* in memory where the data `[1, 2, 3]` *lives*.

The Trick (Zero-Copy):

1. C++ creates a massive array of 768 states in memory.
2. It returns a **pointer** (the address) to Python.
3. NumPy/PyTorch can *wrap* this address and use the data **directly** from that memory location. **No copy is made!**
4. Draw stuff again Timmy!

Concepts for Pipeline 2 (Part B): How Caches *Really* Work

- **CPU Cache:** A tiny, *ultra-fast* memory (L1/L2) on each CPU core. It's ∼100x faster than slow system RAM.
- **The Cache Line (The "64-Byte Chunk")**:
  - When your CPU needs data from *one* memory address (e.g., `rewards[0]`), it doesn't just grab that 1 or 4-byte value.
  - The memory bus fetches the **entire 64-byte "chunk"** of memory that contains it. This is a **Cache Line**.
  - This line is then stored in the core's local cache.
- **Why 768 Environments?**
  - We run 768 environments in parallel, split across 12 CPU threads (1 per core).
  - **768 envs / 12 threads = 64 environments per core.**
  - A cache line is **64 bytes**.
  - This is a perfect match! An array of 64 `bool`s (1 byte each) for one core's worth of environments fits *perfectly* into one 64-byte cache line.

This alignment gives us a massive speedup due to **spatial locality**. And it solves another problem...

Concepts for Pipeline 2 (Part C): C++ Speed Secrets

False Sharing (Very Bad!)

This is a subtle but deadly multi-threading problem.

- **Thread 1** needs to write to `data[0]` so it grabs `[0-64]`.
- **Thread 2** needs to write to `data[1]` so it grabs `[0-64]`.
- When `data[0]` and `data[1]` are next to each other, they might be in the *same cache line* `[0-64]`.
- The CPU cache system gets confused and has to keep "invalidating" the cache for *both* threads, even though they aren't *really* conflicting.
- Both threads stall, and performance is destroyed.

Normal Solution: In C++, we add "padding" to our arrays to *force* Thread 1's data and Thread 2's data into *different* cache lines.
Normal Problem: For this example we use '0' and waste `[1-64]` then cache miss anyway
Our Solution: 768 envs for 12 jobs. 1 byte datatype -> full `[0-64]`
2 byte datatype -> full `[0-64]`, `[64-128]`.

Pipeline 2: The "Vectorized C++" Pipeline

This is the final form: A C++ "data firehose" for the GPU.

- **C++ Backend:**
  - Manages 768 environments in parallel (using **multi-threading**).
  - Writes all `(states, rewards, ...)` into two massive, **contiguous, cache-aligned** numpy arrays (a double buffer).
- **Python Frontend (Main Thread):**
  - Tells C++: "run one step for all 768 envs."
  - C++ returns a **pointer** to the (now full) "inactive" array.
  - Python hands this data (via pointer, no copy) to the GPU.
- **GPU:**
  - Model and Replay Buffer **live permanently on the GPU**.
  - Receives a massive block of 768 transitions at once.
  - Trains on this data in a highly parallel, efficient way.

Result: The C++ backend is a "data factory" at 100% CPU. The GPU is a "training factory" at 100% GPU.

The Results: DEMO TIME

Let's measure performance in **Frames Per Second (FPS)**.

Pipeline	Description	FPS env	Ups/s	Speedup
Cartpole	Basic Gym Loop (Serial)	~ 270	270	1x
Cartpole	Double-Buffered (Parallel)	$\sim 1,900$	270	7x,1x
Multi-Agent	Basic Gym Loop (Serial)	$\sim 1,100$	35	1x
Multi-Agent	Vectorized C++ (768)	$\sim 66,000$	170	$\sim$66x,$\sim$5x

The "NVIDIA" Moment

The vectorized result is why GPUs are so highly valued. This final pipeline *correctly* uses the GPU as a massive parallel processor, which is exactly what it was designed for.

Conclusion & Key Takeaways

1. **Your code is only as fast as its slowest part.** The bottleneck is almost *always* data transfer, not computation.
2. **Parallelize Everything.** Separate data collection (CPU) from model training (GPU). Never let your GPU be idle.
3. **Data Movement is the #1 Cost.** Minimize RAM-to-VRAM transfers. Send data in **large chunks**, not *small streams*.
4. **Use the Right Tool for the Job.** Python for high-level logic, C++ for high-speed simulation, and **Bindings** (like Pybind11) to connect them.
5. **Hardware-Aware > Hardware-Agnostic.** Understanding CPU caches and false sharing is the difference between 1k FPS and 60k+ FPS.

Questions?

Thank You